1990

Heuristic Guided Pre-Optimized Algorithm Substitution for Parallel Computers

Ko-Yang Wang

Report Number:
90-1055

HEURISTIC GUIDED PRE-OPTIMIZED ALGORITHM SUBSTITUTION FOR PARALLEL COMPUTERS

Ko-Yang Wang

CSD-TR-1055
December 1990

Heuristic Guided Pre-Optimized Algorithm Substitution For Parallel Computers†

Ko-Yang Wang

Computing About Physical Objects
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907

ABSTRACT

In this paper, we study the integration of pre-optimized algorithm substitution with the feature-directed program transformation techniques. In the algorithm substitution approach, a pre-optimized algorithm replaces a user program when the pattern of the user program matches that of the pre-optimized algorithm. Parallel program optimization is a highly machine dependent process. However, it is impossible or cost-inefficient to have one pre-optimized algorithm for every architecture. When there is no pre-optimized algorithm for the target machine exists, the feature-directed program optimization process is involved for two purposes: to choose among applicable pre-optimized algorithms or to fine tune the selected pre-optimized algorithm to match the particular architecture that the algorithm is not designed for.

Our solution to the problem of fine-tuning pre-optimized algorithms for different parallel architectures is to record the machine features that the optimization of the algorithm is based upon and use a knowledge base system and a set of program transformations to further restructure the substituted algorithm to fit the target machine.

December 12, 1990

Heuristic Guided Pre-Optimized Algorithm Substitution For Parallel Computers†

Ko-Yang Wang

Computing About Physical Objects
Department of Computer Sciences
Purdue University, West Lafayette, IN 47907

1. Introduction

The program restructuring process maps a program into a functionally equivalent program by altering control or data structures of the program. The goal of the program restructuring is to properly change the order, decomposition, and allocation of control and data structures of the program so that the achievable parallelism can approach the potential parallelism of the program on the target machine.

There are two different approaches to optimize a program on a target machine: the *pre-optimized algorithm substitution* approach and the *program transformation* approach. The *pre-optimized algorithm substitution* approach replaces the program fragment under consideration with an algorithm in the library that has been pre-optimized for the particular target machine. This approach is attractive because there are some widely used numerical algorithms that have been explored extensively on some multiprocessor systems. If these algorithms can be re-used, the expertise and efforts that are spent on optimizing the algorithms can be utilized. The preconditions for this substitution are that the functionalities of the program fragment is the same as those of the pre-optimized algorithm and that the target machine needs to match the machine that the algorithm was optimized for.

The *program transformation* approach improves the match between the program level parallelism and the machine level parallelism in a stepwise refinement process. Each program transformation alters the structure of a program segment slightly to improve the parallelism of the program. It involves techniques that changes the instruction execution order (by forward substitutions, statement reordering, etc), modifying program control (by

† Most material contained in this paper is based on an earlier unpublished paper written in Mar. 1987.

loop interchange, loop distribution, etc), and eliminating unnecessary data accesses and modification (by data localization, block transfer, cache optimization, dead code elimination, etc).

1.1. Pre-optimized Algorithm Substitution vs Program Transformation

The major difference between the pre-optimized algorithm substitution and program transformation approaches lies in the granularity of the changes. The program transformation modifies the program structure step by step, whereas the pre-optimized algorithm substitution takes the "wholesale" approach by replacing the whole program fragment by a pre-optimized algorithm in the library.

The major problem with the pre-optimized algorithm substitution lies in the difficulty of recognizing the opportunity for algorithm substitution. Comparing the semantics of two programs is an NP-complete problem and it is usually very difficult if not impossible for a software system to verify the equivalence of two programs. This approach is useful in interactive programming environments (for example, the FAUST programming environment [GGJMG90]) since the programmer can provide the information interactively that the software system cannot obtain. Another drawback of this approach is that only a small set of pre-optimized algorithms is available in most systems. This is due to the high cost of constructing the library of pre-optimized algorithms. On the other hand, once the opportunity for algorithm substitution is recognized, this approach can achieve very good results since special attention has already been paid to the efficiency of the pre-optimized algorithms. One big advantage of this approach is that the effort done in optimizing some basic arithmetic algorithms can be accumulated and reused.

Optimizing parallel compilers usually takes the program transformation approach, since compilers are particularly well-suited for mechanical analysis of the program dependence and verifying the pre-conditions of the transformations. However, the problem with this approach is that the decision for restructuring the program structure is fragmented. The effect of a transformation on the utilization of the parallelism may not be clear at the time when the transformation is selected. Global analysis of the transformation is needed. Selecting a right sequence of transformations is highly target architecture and program dependent and is very difficult. Finding a generic framework for selecting transformations

is even more difficult and most experts rely on heuristics.

Our opinion is that the two approaches are not necessarily mutually exclusive and they should be combined to makeup the shortcomings of each other.

- At the compiler level, the pre-optimized algorithm substitution can be applied to some well-defined problems, such as those functions defined in BLAS[I,II,III]. Substitutions of more complicated algorithm is possible, but recognizing the opportunity for application is usually too much of a task for the compiler to handle. For programming environments, this requirement is less critical since the user can provide some information to ease the difficulty the environment faces in recognizing the chance for applying pre-optimized algorithms.
- If the opportunity for algorithm substitution is found, it should take precedence over program transformation.
- It is impractical to optimize every useful algorithm for all different target machines. However, parallel program optimization is a highly machine dependent process and the optimizations of algorithms often involve fundamental algorithm changes to utilize the special parallelism features provided by the target machine. Our solution to this problem is as follows:

1. The machine features that the optimization of the algorithm is based upon is recorded in the database along with the algorithm.
2. When there are more than one matching pre-optimized-algorithm exist, select the one that is optimized for a machine that is most close to the target machine by matching the machine features of the two.
3. If the machines in 2. are not exactly the same, use a knowledge base system and a set of program transformations to further restructure the substituted algorithm to fit the target machine.

2. Algorithm Substitution And Fine-Tuning With Pre-Optimized Algorithms

Algorithm substitution approach is attractive because it allows the accumulation of efforts collectively that have been spent on optimizing programs on particular

architectures. Recognizing the opportunities for applying pre-optimized algorithms relies on the pattern matching - matching the pattern of the program dependence graph to the patterns of the pre-optimized algorithms. Selection among the applicable pre-optimized algorithm has to be based on the machine features that the pre-optimized algorithms rely on and the features of the target machine. Fine-tuning a selected pre-optimized algorithm to an architecture that does not match the machine that the algorithm was optimized for involves finding the differences between the architecture features and the consequences of these differences on the optimization of the program. Basing on these observations, the pre-optimized algorithm can be adjusted to the target machine by applying a sequence of program transformations.

Due to the nature of this approach, we will explain the ideas through an example. The example we consider here is the array accumulation problem. This is an interesting problem because the program is simple but it contains data dependences and memory accesses that may serialize the computation. This example allows us to illustrate the heuristics of selecting and fine-tuning the pre-optimized algorithms as well as their applicabilities. It also shows the importance of the resolution of memory contentions. The following is the general form of the array accumulation problems.

```
var
    a : array [1..B] of integer;

for i in 1 .. N do
    a[f(i)] += g(i);
end for
```

In the above program, f is a function that maps the range of the loop index to the range of index of array *a*. For the sake of simplicity, we assume that the accumulation statement is enclosed in a single loop. In more general cases, the accumulation statement may be nested in multiple loops or the array may be a multi-dimensional array.

If the index function f is a one-to-one function, then f is a permutation on a subset of interval [1..B]. In this case, values of g(i) are accumulated into distinct elements of array *a*. When this program is run on a multiprocessor system, no two processors will update

the same memory cell at the same time. However, memory/bus contention problems may still exist because of the limit of bus or network bandwidth, but memory locks are not needed.

For machines with P processing units, one trivial approach is to divide the program into P equal sized tasks and run them on the P processors. For this approach, no pre-optimized algorithm is needed but the speed-up may not be impressive because the efficiency of the approach depends on the values of the index function f, the memory and bus bandwidths as well as the degree of memory interleaving. If more information about the index function is available, then better optimizations are possible.

```
var
        a : array [1..B] of integer;

coprocess i in 1 .. P do
        var s : integer;
        s := N / P;
        for j in (i-1) * s .. i * s - 1  do
                a[f(i)] += g(i);
        end for
end coprocess
```

For example, if the index function is a linear function with respect to the loop indices and the right hand side of the assignment takes about the same amount of time to be processed for all loop instances, then the memory updates may be regulated by updating the memory according to the memory interleaving. This can be accomplished by "index shifting" to shift the index function such that the memory update requests are evenly distributed to all the memory modules. This approach might be the fastest way that we can achieve on some particular machines since the memory updates are processed at their maximum extent.

If f is a constant function then the values of g(i) are accumulated at the same memory cell a[C], where C = f(i). In this case, the problem is called a one bin accumulation problem. If this program is to be executed on a multiprocessor computer with P processors,

each instance of the loop will have to compete to update the same memory cell a[C]. On machines with a combining network that supports the "fetch and add" operations, the accumulation requests can be combined in the network as the requests are routed to the memory. When the requests arrive at the memory module, there will be only one combined memory update request remaining; thus no memory lock is needed. The function call *barrier()* synchronizes the processors and guarantees that each processor starts the next operation at the same time which is essential for the coordination of the "fetch and add" operations.

```
var
        a : array [1..B] of integer;

coprocess i in 1 .. P do
        var s : integer;
                index, value: integer;

        s := N / P;
        for j in (i-1) * s .. i * s - 1  do
                index = f(i);
                value = g(i);
                barrier();
                fetch_and_add(a[index], value);
        end for
end coprocess
```

For machines without a combining network and a "fetch-and-add" operation capability, mutual exclusive accesses to a[C] need to be enforced to avoid memory update conflicts. This might serialize the memory updates and lose all the parallelism of the machine. One possible solution is to block the loop into P chunks and allocate them to P processors; each processor accumulates the values of g(i) in a local counter. These counters are summed up through a tree-sum algorithm after all processors finish their local accumulations. Only one synchronization is needed before the tree-sum operation is started.

```
var
        a : array[1..B] of integer;
        private : array [1..P] of integer;

coprocess i in 1 .. P do
        var s : integer;

        s := N / P;
        private[i] := 0;

        for j in (i-1) * s .. i * s - 1  do
                private[i] += g(j);
        end for
end coprocess

barrier();
a[f(1)] := tree_sum(private[*]);
```

If the expression g(j) contains no function calls then the private accumulations can be computed in N/P cycles and the tree-sum algorithm can be computed in O(log P) cycles. Therefore, the overhead of the tree-sum operation is O(N/P + log P) which is constant with respect to the number of processors P in the machine.

If the function f is neither a constant function nor a one-to-one function, this problem is called a multiple bins accumulation problem because the values of g(i) are accumulated into many elements of array *a* simultaneously. A significant number of synchronization is needed because of the unpredicability of the values of the function f. Processors may compete to update any of the array elements at any instance, so memory locks need to be placed on all elements of the array *a* to guard correct memory updates.

All these three kinds of array accumulation problems are common in practical applications. One particularly interesting example of these is the image processing algorithm called "histogramming." The histogramming problem is a special case of the multiple

bins accumulation problem whose index function is array reference. In image processing, a picture frame is represented by a two dimensional array of points called *pixels*. Each *pixel* has a small value between 0..(b-1) (typically 8-bit numbers) that represent the grey scale value or the color RGB value of the point. The histogramming involves keeping track of the occurrence of each grey scale value in the picture.

The following sequential program accumulates the occurrences of each grey scale value of the picture represented by the array *pixel*.

```
        function histogram(pixel, m, n, histog, b)
          variable
            pixel  : array [0 .. m-1, 0 .. n-1] of integer;
            histog : array [0 .. b-1] of integer;

        histog := 0;    /* initialize the counter array to zero */

        for i in 0 .. m-1 do
            for j in 0 .. n-1 do
S1:             histog[pixel[i,j]] := histog[pixel[i,j]] + 1;
            end for
        end for
```

The histogram helps to distinguish regions which are similar in some respects. For example, the region of *pixels* that have gray scale values between certain limits may indicate that the area of the picture is the sky or a particular object with certain color. Subsequent filtering on the basis of the limits provided by the histogram may isolate the region of interest. This example was used in [HoRe77] to illustrate the techniques of hand decomposing the problem for three differently configured multiprocessor systems.

Throughout this section, the multiple bins accumulation problem is used to demonstrate the general ideas, but at times when more detail illustration is needed the histogramming problem will be used.

The multiple bins problem can be detected by the system by matching the program with the general form of the multiple bin described above. If the index function f involves

only loops indices, then the system can detect the type of the problem in hand by applying array subscript tests. However, when there are variables other than the loop indices involved, telling whether the index function is a constant function or a one-to-one function may not be trivial for the system. For example, the index may be an array reference whose i-th element has value i. Although it may be trivial to the user that this index function is an identity function, the system will not be able to sense this fact simply by the static analysis. In these cases, the system will have to rely on the user to provide help through user interactions. If the subscript test fails and the user cannot provide help, a heuristic is applied and the more complicated multiple bin problem is assumed.

When a multiple bins accumulation problem is recognized by the system, pre-optimized algorithms for this problem are considered. As we explained in section 3, the computational model needs to be analyzed to see if the pre-optimized algorithm is suitable for the architecture. Heuristics are also considered in fine-tuning the pre-optimized algorithm to match with the computational model.

For machines that support combining networks and "fetch and add" operations, the accumulation operations in the multiple bins accumulation problem can be translated into "fetch and add" operations. If we divide the program into P tasks by loop blocking and run the tasks by P processors, there will be at most P "fetch and add" requests for the same memory location at each cycle. Requests that have the same destinations are combined in the network and merged into one request when arriving at the memory cell. The "fetch and add" operations eliminate the need for memory locks so the overall performance of the algorithm will be improved. However, the speed-up of this approach may not be significant, because there may be many memory cells to be referenced at the same operation cycle. Depending on the reference patterns, the memory update requests may pile up in the network and memory modules may thus cause network saturation.

The two-phase private counters approach we used in the one-bin accumulation problem can be extended to solve the multiple bins accumulation problem on multiprocessor machines that have several memory modules. During the first phase, an array of private accumulation counters is created for each processor that runs the program. Each processor gets a share of the job and updates the private counters independently. In the second phase, these private counters are summed up either by tree-sum or other available parallel

summation algorithms.

The memory updates of a[f(indices)] in the original program are irregular and unpredictable. In the first phase of the two-phase approach, the memory access pattern of private_a is still unpredictable. But, since each processor exclusively owns its private counters, the private counters can be updated simultaneously and independently. No cache coherence problem needs to be worried about. One possible problem is that when the machine has a small cache, the irregular updates of the array private_a may produce a high cache miss ratio. However, in most problems, the private counters are fairly small. For example, the private counter array in the histogramming problem is of size 2**b. When b is equal to 8, the size of the private counter array is 256 so the entire array can reside in the cache throughout phase 1.

On the other hand, the memory update pattern in phase 2 is very regular. Very few synchronization is needed. Processors can cooperate to sum up the private counters. On most machines, the pre-defined tree-sum algorithm will provide a reasonably good speed-up.

Note that this approach of introducing private counters and dividing the memory accesses into two phases is based on the expertise about this particular array accumulation problem and may not be applicable to other problems. Therefore, the pre-optimized algorithm is used. The pre-optimized algorithm did not specify where the private accumulation counters should be allocated because this problem falls into the category of the array allocation problem and the general heuristics about array allocation can be used to decide how the private counters should be allocated. In general, not all the details in pre-optimized algorithms need to be implemented because some of them are covered by general heuristics. As a result, some of the variations of the pre-optimized algorithm can be left undetermined until the actual program is substituted. The unspecified part can then be obtained by applying the transformations. This approach cuts down the number of pre-optimized algorithms that the system needs to store significantly.

```
var
        a : array[1..B] of int
        private_a: array [1..P, 1..B] of integer;
        s : integer;

s := N / P;
coprocess i in 1 .. P do
        for j in (i-1) * s .. i * s - 1  do
                private_a[i, f(j)] += g(j);
        end for
end coprocess

barrier();
for j in 1 .. B do
        a[j] := tree_sum(private_a[*, j]);
end for;
```

The implementation of the function tree_sum() may vary from machine to machine. For machines that have combined networks, the summations can be accomplished by merging the "fetch and add" requests in the network. In this case, the tree-sum loop can be translated into :

```
coprocess i in 1 .. P
        for j in 1 .. B do
                fetch_and_add(a[j], private_a[i, j]);
        end for
end coprocess
```

At each cycle, the P processes will generate P "fetch and add" requests to the same memory location. These requests are combined in the network and merged into one request when arriving at the memory. Since each processor accesses the same memory location at the same cycle, no hot spot network saturation problem will ever occur. This last statement will be valid only when all the processors are homogeneous and the system

provides a barrier synchronization routine that can start all processors at the same time

On other machines that do not support "fetch and add" operations, the tree-sum operations can be handled by simulating the binary tree network. We can parallelize the outermost loop by synchronizing the memory accesses of the tree-sum operations.

```
for pid in P .. 1 step -1 do
   local private_a: array [1..P, 1..B] of integer;
   for i in 1 .. B do
      if (pid == 1) then
         a[i] := private_a[pid,i] + private_a[pid*2,i] + private_a[pid*2+1,i];
      else if (pid <= P / 2) then      /* is not a leaf */
         private_a[pid,i} += private_a[pid*2,i] + private_a[pid*2+1,i];
      end if
   end for
end for
```

In the above program, there are two pair of dependences from private_a[pid*2, i] and private_a[pid*2+1, i] to private_a[pid, i]. These dependences correspond to the inherent characteristic of the tree operations: the operations in the intermediate nodes of the tree cannot be executed until the children of the node finish the computations. If the array private_a is allocated in the shared memory then semaphores need to be inserted to enforce the order of the tree computations. In the Blaze language, the synchronization primitives hide the distinctions between synchronization variables and inter-processor channel variables in implementation. To avoid confusing the readers by the Blaze synchronization variables with the channel variables we discussed above, we use the more conventional semaphore notations *signal()* and *wait()* in the following program:

```
var
   syn: array[1..P] of semaphore;

coprocess pid in P .. 1 step -1 do
   local private_a: array [1..P, 1..B] of integer;
   for i in 1 .. B do
      if (pid == 1) then
        a[i] := private_a[pid,i] + private_a[pid*2,i] + private_a[pid*2+1,i];
      else
         if (pid <= P / 2) then     /* is not a leaf */
            wait(syn[pid*2]);   wait(syn[pid*2+1]);
            private_a[pid,i] += private_a[pid*2,i] + private_a[pid*2+1,i];
         end if;
         signal(syn[pid/2]);
      end if;
   end for;
end coprocess;
```

If the array private_a is allocated to the local memories then the values can be passed to other processors by either copying them to shared memory with synchronization locks or sending them through the inter-processor communication channels.

For a non-shared memory machine like Pringle, the "tree-sum" operations are pipelined through a tree configuration of the machine. The switches in the network are configured such that the processor with processor id pid is connected to its parent whose id is pid/2 and its two children whose processor ids are pid*2 and pid*2+1. The processor with processor id 1 is the root of the tree. The synchronization and the buffers are hidden in the implementation of the channel variables on the machine.

The pipelined tree-sum algorithms can be executed in time C * log P * B where C is a small constant.

```
for i in 1 .. B do
    if (is_leaf(pid)) then
        CH_parent <- private_a[pid, i];
    else
        l <- CH_l_child;  r <- CH_r_child
        CH_parent <- l + r + private_a[pid, i];
    end if
end for
```

For distributed-memory architectures that have non-trivial communication overheads (such as NCUBE/2 and iPSC/2), the pipe-line approach for the tree-sum on them is prohibitedly expensive. For these machines the tree-sum operation is performed in blocks of vectors to minimize the communication overhead.

To summarize, the unpredictable memory access pattern of this problem makes the loop optimization techniques ineffective no matter how the control structure is modified. Creating private accumulation counters regulates the memory update patterns and eliminates the need for memory locking. The tradeoff is that more memory cells and computing cycles are used to store and sum up the private counters. These costs are constant with respect to the number of processors P and the size of the array *a* B. Also, these costs may be compensated for by minimizing synchronization and resolving memory contentions.

For problems like the histogramming program, the memory accesses pattern is highly data dependent. There is no easy way for the compiler to tell whether the extra cost of manipulating the private bins justifies the synchronization costs it saves. Heuristics are used in deciding whether pre-optimized algorithms are suitable and beneficial. Use of the heuristics also allows the compiler to be more aggressive in parallelizing the program.

3. Conclusions

In this paper, the use of pre-optimized algorithms is demonstrated; variations of the algorithms are used for different architecture configurations. The choice of the variations in implementation is determined by a set of rules based on the computational model. After the algorithm substitution, basic transformations are applied to match the algorithm

with the underlying architecture better.

A pre-optimized algorithm may utilize other pre-defined algorithms in the process of fine-tuning. For example, the two-phase approach in solving the multiple-bin accumulation problem uses the pre-optimized pipelined tree-sum algorithm. The actual implementations of the tree-sum algorithm can be based on a finer classification of parallel architectures. This approach allows the specification of generic pre-optimized algorithms that may be used for a wider range of parallel architectures.

The pre-optimized algorithm approach differs from the fine grain heuristics in the level of the heuristics: the pre-optimized algorithms are pre-packed special purpose heuristics which are only suitable for the particular problem they are designed for. Better optimizations may be achieved with pre-optimized algorithm because the particular problems are considered in detail.

4. REFERENCES

[ABCCF88] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante, "An Overview of the PTRAN analysis system for Multiprocessing," in *Proceedings of the 1987 International Conference on Supercomputing, LNCS, February, 1988.*

[AlBaKe86] J.R. Allen, D. Baumgartner, K. Kennedy, A. Porterfield, "PTOOL: A Semi-Automatic Parallel Programming Assistant," *1986 International Conference on Parallel Processing,* August, 1986, 164-170.

[AlKe84] J.R. Allen, and K. Kennedy, "A Parallel Programming Environment," technical report, Rice COMP TR84-3, July 1984.

[BuCy86] M. Burke, R. Cytron, "Interprocedural Dependence Analysis and Parallelization," *SIGPLAN symposium on Compiler Construction,* 1986, 613-641.

[CCHKT88] C.D. Callahan, K.D. Cooper, R.T. Hood, K. Kennedy, and L. Torczon, "Parascope: A Parallel Programming Environment," *The International Journal of Supercomputer Applications,* Vol 2. No. 4, Winter, 1988, pp. 84-99.

[GGJMG89] V. Guarna Jr., D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur, "Faust: an Integrated Environment for the development of Parallel Programs," *IEEE software,* July 1989.

[HoRe77] R. Hon, and D.R. Reddy, "The Effects of Computer Architecture on Algorithm Decomposition and Performance," in *High-Speed Computers and Algorithm Organization* Kuck, et al., editors, Academic Press, 1977, 411-421.

[HwBr84] K. Hwang and F. Briggs, "Computer Architecture and Parallel Processing," McGraw-Hill, 1984.

[PGHLLS89]C.D. Polychronopoulos, M. Girkar, M.R. Haghighat, C.L. Lee, B. Leung, and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors," in Conference Proceeding, Int'l Conf. on Parallel Processing, 1989.

[Wang89] K. Wang, "Machine Knowledge Representation and Manipulation For Parallel Compilers," Technical Report CSD-TR-843, Department of Computer Sciences, Purdue University, Jan. 1989.

[WaGa89] K. Wang and D. Gannon, "Applying AI Techniques to Program Optimizations For Parallel Computers," in K. Hwang and D. DeGroot, editors, *Parallel Processing for Supercomputers and Artificial Intelligence*, McGraw-Hill, 1989, pp. 441-485.

[Wang90a] K. Wang "A Performance Predication Model For Parallel Compilers," Tech. Report, CSD-TR-1041, CER-90-43, Department of Computer Science, Purdue University, Nov. 1990.

[Wang90b] K. Wang "Managing Data Synchronization Automatically For Distributed-Memory Architectures," Tech. Report, CSD-TR-1043, CER-90-45, Department of Computer Science, Purdue University, Nov. 1990.

[Wang90c] K. Wang "A framework For Intelligent Parallel Compilers," Tech. Report, CSD-TR-1044, Department of Computer Science, Purdue University, Nov. 1990.